How do you compute the midpoint of an interval?

Frédéric Goualard

A

How do you compute the midpoint of an interval?

FRÉDÉRIC GOUALARD, CNRS, LINA, UMR 6241

The algorithm that computes the midpoint of an interval with floating-point bounds requires some careful devising to handle all possible inputs correctly. We review several implementations from prominent C/C++ interval arithmetic packages and analyze their potential failure to deliver the expected results. We then show how to amend them to avoid common pitfalls. The results presented are also relevant to non-interval arithmetic computation such as the implementation of bisection methods. Enough background on IEEE 754 floating-point arithmetic is provided for this paper to serve as a practical introduction to the analysis of floating-point computation.

Categories and Subject Descriptors: G.1.0 [**General**]: Computer arithmetic; Error analysis; Interval arithmetic

General Terms: Algorithms, Reliability, Experimentation

Additional Key Words and Phrases: floating-point number, IEEE 754 standard, interval arithmetic, midpoint, rounding error

1. INTRODUCTION

In his 1966 report [Forsythe 1966] "*How do you solve a quadratic equation?*", Forsythe considers the seemingly simple problem of reliably solving a quadratic equation on a computer using floating-point arithmetic. Forsythe's goal is both to warn a large audience away from unstable classical textbook formulae as well as get them acquainted with the characteristics of pre-IEEE 754 standard floating-point arithmetic, a dual objective shared by his later paper "*Pitfalls in Computation, or Why a Math Book isn't Enough*"[Forsythe 1970].

Following Forsythe's track, we consider here the problem of computing a good approximation to the midpoint between two floating-point numbers. We strive to provide both a reliable algorithm for midpoint computation and an introduction to floating-point computation according to the IEEE 754 standard [IEEE 1985][1].

Given two real numbers a and b from $\mathbb{R}$, with $a \leqslant b$, the midpoint $\mathsf{m}(\boldsymbol{I})$ of the closed interval $\boldsymbol{I} = [a, b] = \{x \in \mathbb{R} \mid a \leqslant x \leqslant b\}$ can easily be obtained with the straightforward formula:

$$\mathsf{m}([a, b]) = \frac{a + b}{2}. \tag{1}$$

[1] A new version of the original standard from 1985 was approved and released in 2008 [IEEE 2008]. Most prominently, it considers both binary and decimal representations of floating-point numbers, while the 1985 version only considered a binary representation. In this paper, we only consider the features standardized by the 1985 version—by far the most used to this day—for the sake of simplicity.

Part of the work presented here was funded by IFCPAR/CEFIPRA Project 4502-1.
Author's address: F. Goualard, Université de Nantes. Nantes Atlantique Université. CNRS, LINA, UMR 6241, 2 rue de la Houssinière, BP 92208, F-44322 NANTES CEDEX 3.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

But, if a and b are two *floating-point numbers* from a set $\mathbb{F}$ of *finite* floating-point numbers, the sum $a+b$ may not be a floating-point number itself, and we therefore have to take care of rounding it correctly to ensure that our floating-point implementation of $\mathsf{m}(\boldsymbol{I})$ does not violate the fundamental property:

$$\mathsf{m}(\boldsymbol{I}) \in \boldsymbol{I}, \tag{2}$$

viz., that the floating-point midpoint of an interval $\boldsymbol{I}$ should belong to $\boldsymbol{I}$.

This property is a prerequisite to use the midpoint operator in the dichotomic search process of some numerical algorithm, even though it is not sufficient to ensure that the search space is separated into sub-spaces of the same size[2]. There are, however, problems that require more from a midpoint operator, *viz.*, that it computes the floating-point number closest to the real midpoint of $\boldsymbol{I}$[3]:

$$\forall v \in \mathbb{F}: \quad |c - v| \geqslant |c - \mathsf{m}(\boldsymbol{I})|\,, \quad \text{with } c = \frac{a+b}{2}, c \in \mathbb{R}. \tag{3}$$

In interval arithmetic [Moore 1966; Alefeld and Herzberger 1983; Hayes 2003], some examples of such problems are:

— On average, the least overestimation in defining a *centered form* [Moore 1966; Ratschek 1980] of a rational function is obtained by choosing a value closest to the midpoint as developing point [Ratschek and Rokne 1984, p. 41];
— For the Krawczyk operator [Neumaier 1990, p. 177], the midpoint is the value that gives the tightest result over all other points of the interval under consideration [Neumaier 1990, Theo. 5.1.9, p. 178].

Inclusion and precision are not, however, the only possible problems for floating-point implementations of the midpoint operator: what should be, for example, the midpoint of an empty interval? As we shall see in Section 2, the IEEE 754 standard comes to the rescue here by defining a special *Not-a-Number* value (NaN) that should be used for such situation; another difficulty lies in the fact that, as we will see in the next section, one or both bounds of an interval with floating-point bounds may be an infinite value "$\pm\infty$". What should then be the midpoint of an interval of the form $[a, +\infty]$, $[-\infty, b]$, or even $[-\infty, +\infty]$? Though the midpoint is mathematically undefined, many applications (*e.g.*, constraint solvers such as IBEX [Chabert et al. 2012] or Realpaver [Granvilliers and Benhamou 2006]) typically expect finite values and would behave incorrectly, should the midpoint operator return a NaN for these intervals. As a consequence, the current draft of the future IEEE P1788 standard on interval arithmetic defines the midpoint operator as follows:

$$\begin{cases} \mathsf{m}(\varnothing) & \text{is NaN}, \\ \mathsf{m}([-\infty, b]) & = -\texttt{realmax}, \\ \mathsf{m}([a, +\infty]) & = \texttt{realmax}, \\ \mathsf{m}([-\infty, +\infty]) & = 0, \\ \mathsf{m}([a, b]) & = \mathsf{rndnr}\langle (a+b)/2 \rangle\,, \end{cases} \quad \text{for } (a, b) \in \mathbb{F}^2, \tag{4}$$

where $\mathsf{rndnr}\langle x \rangle$ is defined in the next section as returning the floating-point value closest to the real number x, and `realmax` is the largest positive element of $\mathbb{F}$.

For non-empty intervals with finite bounds, Equation (3) implies Equation (2). However, some implementations may choose to ensure only the latter and a relaxation of the former.

Due to its many uses (centered forms, Krawczyk operator, Newton operator [Moore 1966], computation of preconditioning matrices [Kearfott 1990], representation of intervals in the

[2]Where the size $\mathsf{w}(\boldsymbol{I})$ of an interval $\boldsymbol{I} = [a, b]$ with floating-point bounds is defined as $\mathsf{w}(\boldsymbol{I}) = b - a$.
[3]If the real midpoint is midway between two floating-point numbers, there exist two possible values for $\mathsf{m}(\boldsymbol{I})$ that satisfy Equation (3).

midpoint/radius format [Rump 1999a], ...), the midpoint operator is a staple of interval arithmetic libraries. It is, therefore, paramount that its floating-point implementation at least satisfies Equation (2). Accuracy, as stipulated by Equation (3), is also desirable, as seen in the examples above. Nevertheless, we will see in Section 3 that some formulae implemented in popular C/C++ interval libraries may not ensure even the containment requirement for some inputs. However, this study should not be considered as a report on the quality of these libraries since different implementations of the midpoint operator might suit specific needs, and some libraries were defined with no provision to support intervals with infinite bounds in the first place. We are only concerned here with the properties of the formulae these libraries implement.

In Section 3, we analyze the various formulae both theoretically and practically; contrary to most expositions, we consider the impact of both overflow and underflow on the accuracy and correctness of the formulae.

The error analysis conducted in this paper requires slightly more than a simple working knowledge of floating-point arithmetic as defined by the IEEE 754 standard. As a consequence, the basic facts on floating-point arithmetic required in Section 3 are presented in Section 2 for the sake of self-containedness.

It turns out that the study of the correct implementation of a floating-point midpoint operator may serve as a nice introduction to many important aspects of floating-point computation at large: the formulae studied are simple enough for their analysis to be easily understandable, while the set of problems raised is sufficiently broad in scope to be of general interest. We then hope that this paper will be valuable as both a technical presentation of reliable, accurate, and fast methods to compute a midpoint as well as an introduction to error analysis of floating-point formulae.

2. FLOATING-POINT ARITHMETIC IN A NUTSHELL

According to the IEEE 754 standard [IEEE 1985], a floating-point number φ is represented by a sign bit s, a significand m (where m is a bit string of the form "$0.f$" or "$1.f$", with f the *fractional part*) and an integral exponent E:

$$\varphi = (-1)^s \times m \times 2^E. \tag{5}$$

The IEEE 754 standard defines several formats varying in the number of bits $l(f)$ and $l(E)$ allotted to the representation of f and E, the most prominent ones being *single precision*—$\big(l(E), l(f)\big) = \big(8, 23\big)$—and *double precision*—$\big(l(E), l(f)\big) = \big(11, 52\big)$. We will also use for pedagogical purposes an *ad hoc* IEEE 754 standard-compliant *tiny* precision format—$\big(l(E), l(f)\big) = \big(3, 3\big)$.

Wherever possible, the significand must be of the form "$1.f$" since it is the form that stores the largest number of significant figures for a given size of m:

$$\begin{aligned} \varphi &= \mathbf{0.011}01 \times 2^{0} \\ &= \mathbf{0.110}1 \times 2^{-1} \\ &= \mathbf{1.101} \times 2^{-2}. \end{aligned}$$

Floating-point numbers with such a significand are called *normal numbers*. Such prevalence is given to normal numbers that the leading "1" is left implicit in the representation of an IEEE 754 number, and only the fractional part f is stored (see Figure 1).

The exponent E is a signed integer stored as a biased exponent $e = E + \text{bias}$, with $\text{bias} = 2^{l(E)-1} - 1$. The biased exponent e is a non-negative integer that ranges from $e_{\min} = 0$ to $e_{\max} = 2^{l(E)} - 1$. However, for the representation of normal numbers, E only ranges from $E_{\min} = (e_{\min} - \text{bias}) + 1$ to $E_{\max} = \big(e_{\max} - \text{bias}\big) - 1$ because the smallest and largest values are reserved for special purposes (see below). As an example of what

precedes, the bias for the *tiny* format is equal to 3, e ranges from 0 to 7, and E ranges from -2 to $+3$.

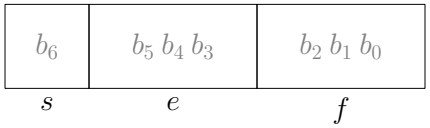


Fig. 1. Binary representation as a seven bit string of a *tiny* floating-point number.

Consider the binary number $\rho = 1.0011$. It cannot be represented as a *tiny* floating-point number since its fractional part has four bits, and the *tiny* format has room for only three. It therefore has to be rounded to the floating-point number $\mathsf{fl}\langle\rho\rangle$ according to one of four rounding directions[4] (see Figure 2):

— Rounding toward 0: $\mathsf{fl}\langle\rho\rangle = \mathsf{rndzr}\langle\rho\rangle$;
— Rounding to nearest-even: $\mathsf{fl}\langle\rho\rangle = \mathsf{rndnr}\langle\rho\rangle$;
— Rounding toward $-\infty$: $\mathsf{fl}\langle\rho\rangle = \mathsf{rnddn}\langle\rho\rangle$;
— Rounding toward $+\infty$: $\mathsf{fl}\langle\rho\rangle = \mathsf{rndup}\langle\rho\rangle$.

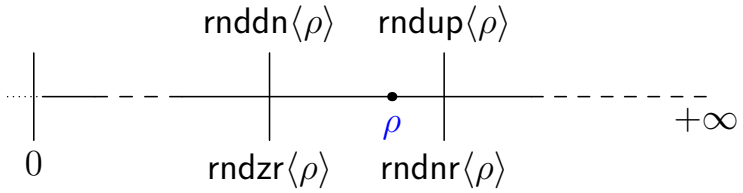


Fig. 2. Rounding a real number according to the IEEE 754 standard.

Note the use of angles "⟨⟩" instead of more conventional parentheses for the rounding operators. They are used to express the fact that each individual value and/or operation comprising the expression is individually rounded according to the leading operator. For example:

$$\mathsf{fl}\langle\rho_1 + \rho_2\rangle \equiv \mathsf{fl}(\mathsf{fl}(\rho_1) + \mathsf{fl}(\rho_2)), \quad \forall(\rho_1, \rho_2) \in \mathbb{R}^2.$$

When rounding to nearest, if ρ is equidistant from two consecutive floating-point numbers, it is rounded to the one whose rightmost bit of the fractional part is zero (the "even" number).

It is not possible to represent 0 as a normal number. Additionally, consider the number φ:

$$\varphi = 0.00011 \times 2^0.$$

To store it as a *tiny* floating-point normal number requires shifting the leftmost "1" of the fractional part to the left of the radix point:

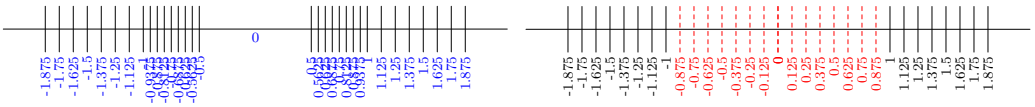
$$\varphi = 1.100 \times 2^{-4}.$$

[4]The actual direction chosen may depend on the settings of the Floating-Point Unit at the time, or alternatively, on the machine instruction used, for some architectures.

However, doing so requires an exponent smaller than $E_{\min}$. It is nevertheless possible to represent φ, provided we accept to store it with a "0" to the left of the radix point:

$$\varphi = 0.011 \times 2^{-2}.$$

Numbers with a significand of the form "$0.f$" (with $f \neq 0$) are called *subnormal numbers*. Their introduction is necessary to reduce the large gap that would otherwise occur around 0 (compare Figure 3(a) and Figure 3(b)). The value "0" is also represented with the form "$0.f$" (with $f = 0$) but it is considered by the IEEE 754 standard as neither normal nor subnormal.



(a) Without subnormal numbers. (b) With subnormal numbers (dashed red).

Fig. 3. The *tiny* floating-point format with and without subnormals (focus on 0).

To signal that a number is a subnormal number, the biased exponent e stored is set to the reserved value 0, even though the unbiased exponent E is $E_{\min}$ (otherwise, it would not be possible to distinguish between a normal number and a subnormal number whose unbiased exponent is $E_{\min}$). Exceptional steps must be taken to handle subnormal numbers correctly since their leading bit is "0", not "1". This has far reaching consequences in terms of performance, as we will see at the end of Section 3.

Figure 4 shows the repeated division by two of a *tiny* number across the normal/subnormal divide. As seen in that figure, dividing an IEEE 754 floating-point number by two is an error-free operation if the result is not a subnormal number. Otherwise, the rightmost bit of the fractional part is shifted out, and if it is non-zero, the operation is not error-free. The values λ and μ are, respectively, the *smallest positive normal* and the *smallest positive subnormal* floating-point numbers.

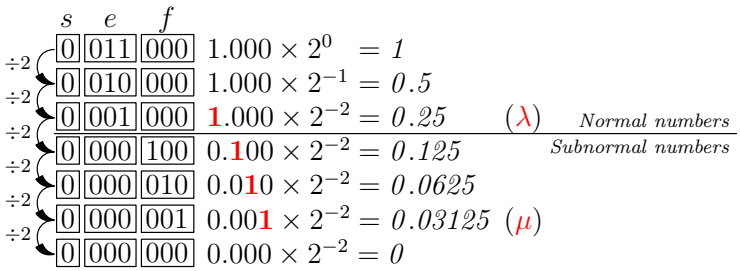


Fig. 4. Repeated division by two from 1.0 to 0.0 in the *tiny* format (rounding to nearest/even).

The condition whereby an operation leads to the production of a subnormal number or an inexact zero is called *underflow*. On the other side of the spectrum, an operation may produce a number that is too large to be represented in the floating-point format on hand, potentially leading to an *overflow*. In that case, depending on the rounding policy in effect, the number is replaced either by an *infinity* of the right sign—which is a special floating-point number whose biased exponent e is set to $e_{\max}$ and whose fractional part is set to

zero—, or by the largest (in absolute value) finite floating-point number with the right sign[5]. In the rest of this paper, we denote by $\mathbb{F}$ the set of normal and subnormal floating-point numbers and by $\overline{\mathbb{F}} = \mathbb{F} \cup \{-\infty, +\infty\}$ its affine extension, the size of the underlying format (*tiny* or *double* precision, mainly) being unambiguously drawn from the context.

To ensure non-stop computation even in the face of a meaningless operation, the IEEE 754 standard defines the outcome of all operations, the undefined ones generating a NaN (*Not a Number*), which is a floating-point datum whose biased exponent is set to $e_{\max}$ and whose fractional part is any value different from zero. We will have, *e.g.*:

$$\sqrt{-1} = \text{NaN}, \quad \infty - \infty = \text{NaN}.$$

The NaNs are supposed to be unordered, and any test in which they take part is false[6]. As a consequence, the right way to test whether an interval $[a, b]$ is empty is to check whether $\neg(a \leqslant b)$ is true since that form returns a correct result even if either a or b is a NaN.

To sum up, the interpretation of a bit string representing a floating-point number depends on the values of e and f as follows:

$$\begin{cases} e = 0, & f = 0: & \varphi = (-1)^s \times 0.0 \\ e = 0, & f \neq 0: & \varphi = (-1)^s \times (0.f) \times 2^{1-\text{bias}} \\ 0 < e < e_{\max} & : & \varphi = (-1)^s \times (1.f) \times 2^{e-\text{bias}} \\ e = e_{\max} & f = 0: & \varphi = (-1)^s \times \infty \\ e = e_{\max} & f \neq 0: & \varphi = \text{NaN}. \end{cases}$$

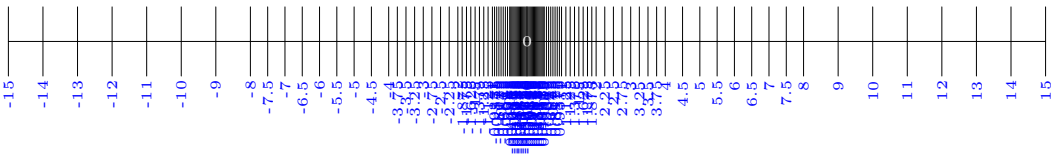


Fig. 5. IEEE 754 *tiny* floating-point (normal and subnormal) numbers.

Figure 5 presents the distribution of all *tiny* numbers from $\mathbb{F}$ on the real line. The farther from 0, the larger the gap from a floating-point number to the next. More specifically, as Figure 6 shows, the difference between two consecutive floating-point numbers doubles every $2^{l(e)}$ numbers in the normal range, while it is constant throughout the whole subnormal range.

Rounding is order-preserving (*monotonicity of rounding*):

$$\forall(\rho_1, \rho_2) \in \mathbb{R}^2 \colon \rho_1 \leqslant \rho_2 \implies \mathsf{fl}\langle\rho_1\rangle \leqslant \mathsf{fl}\langle\rho_2\rangle$$

for the same instantiation of $\mathsf{fl}\langle\rangle$ in both occurrences. The IEEE 754 standard also mandates that real constants and the result of some operations (addition, subtraction, multiplication, and division, among others) be *correctly rounded*: depending on the current rounding direction, the floating-point number used to represent a real value must be one of the two floating-point numbers surrounding it. Disregarding overflow—for which error analysis is not very useful—, a simple error model may be derived from this property: let ρ be a positive real number[7], φ_l the largest floating-point number smaller or equal to ρ and φ_r the smallest floating-point number greater or equal to ρ. Let also $\mathsf{rndnr}\langle\rho\rangle$ be the correctly

[5]For example, if the sum $a+b$ overflows with $a+b > 0$, we have $\mathsf{rnddn}\langle a+b\rangle = \texttt{realmax}$ and $\mathsf{rndup}\langle a+b\rangle = +\infty$.

[6]Surprisingly enough to the unsuspecting, even the equality test turns false for a NaN, so much so that the statement $x \neq x$ is an easy way to check whether x is a NaN.

[7]The negative case may be handled analogously.

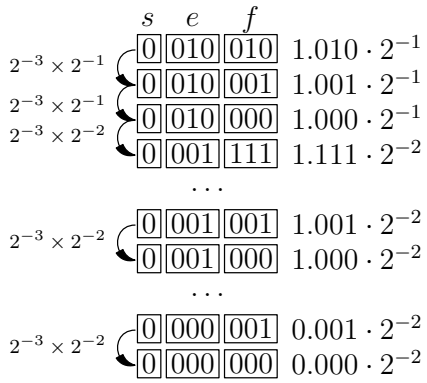


Fig. 6. Gap between consecutive floating-point numbers in the *tiny* format.

rounded representation to the nearest-even of ρ by a floating-point number ($\mathsf{rndnr}\langle\rho\rangle$ is either φ_l or φ_r). If ρ is a floating-point number, we have:

$$\mathsf{rndnr}\langle\rho\rangle = \rho. \tag{6}$$

Otherwise, we have:

$$|\mathsf{rndnr}\langle\rho\rangle - \rho| \leqslant \frac{\varphi_r - \varphi_l}{2}. \tag{7}$$

We may consider two cases depending on whether φ_l is normal or not:

— *Case φ_l normal.* The real ρ may be expressed as $m_\rho \times 2^E$, with $1 \leqslant m_\rho < 2$. Then, φ_l can be put into the form $m_{\varphi_l} \times 2^E$, with $1 \leqslant m_{\varphi_l} < 2$. If we call ε_M the *machine epsilon* corresponding to the value $2^{-l(f)}$ of the last bit in the fractional part[8] of a floating-point number, we have $\varphi_r = \varphi_l + \varepsilon_M \times 2^E$. From Equation (7), we get:

$$|\mathsf{rndnr}\langle\rho\rangle - \rho| \leqslant \frac{\varepsilon_M}{2} \times 2^E.$$

We have[9]:

$$\frac{|\mathsf{rndnr}\langle\rho\rangle - \rho|}{\rho} \leqslant \frac{(\varepsilon_M/2) \times 2^E}{m_\rho \times 2^E}.$$

We know $1 \leqslant m_\rho < 2$. Then:

$$|\mathsf{rndnr}\langle\rho\rangle - \rho| \leqslant \frac{\varepsilon_M}{2}\rho$$

or, alternatively:

$$\mathsf{rndnr}\langle\rho\rangle = \rho(1+\delta), \quad \text{with } |\delta| \leqslant \frac{\varepsilon_M}{2}. \tag{8}$$

— *Case φ_l subnormal.* The distance between any subnormal number to the next floating-point number is constant and equal to $\mu = \varepsilon_M \times 2^{E_{\min}}$. From Equation (7), we then have:

$$|\mathsf{rndnr}\langle\rho\rangle - \rho| \leqslant \frac{\mu}{2},$$

[8] Alternatively, ε_M may be defined as the difference between 1.0 and the next larger floating-point number.
[9] Here, ρ is different from 0 since it is not a floating-point number at this point.

which may be expressed as:

$$\mathsf{rndnr}\langle\rho\rangle = \rho + \eta, \quad \text{with } |\eta| \leqslant \frac{\mu}{2}. \tag{9}$$

We may unify Equation (6), Equation (8), and Equation (9) as:

$$\mathsf{rndnr}\langle\rho\rangle = \rho(1+\delta) + \eta, \quad \text{with } \begin{cases} \delta\eta = 0, \\ |\delta| \leqslant \frac{\varepsilon_M}{2}, \\ |\eta| \leqslant \frac{\mu}{2}, \end{cases} \tag{10}$$

where either one of δ and η has to be null ($\delta = 0$ in case of underflow, and $\eta = 0$ otherwise). This error model is valid for all correctly rounded operations and then:

$$\forall(\varphi_1, \varphi_2) \in \mathbb{F}^2,\ \forall \top \in \{+, -, \times, \div\}: \\ \mathsf{rndnr}\langle\varphi_1 \top \varphi_2\rangle = (\varphi_1 \top \varphi_2)(1+\delta) + \eta, \quad \text{with } \begin{cases} \delta\eta = 0, \\ |\delta| \leqslant \frac{\varepsilon_M}{2} \\ |\eta| \leqslant \frac{\mu}{2}. \end{cases} \tag{11}$$

For the other rounding directions, the same error model can be used, except that the bounds on δ and η are now larger:

$$|\delta| < \varepsilon_M \text{ and } |\eta| < \mu.$$

Table I. Characteristics of floating-point formats.

Format	$l(E)$	$l(f)$	$E_{\min}$	$E_{\max}$	$\varepsilon_M^{\dagger}$	$\lambda^{\star}$	$\mu^{\diamond}$
tiny	3	3	-2	$+3$	2^{-3}	2^{-2}	2^{-5}
single	8	23	-126	$+127$	2^{-23}	2^{-126}	2^{-149}
double	11	52	-1022	$+1023$	2^{-52}	2^{-1022}	2^{-1074}

†) *machine epsilon*, equal to $2^{-l(f)}$
⋆) ***realmin*** (smallest positive normal number), equal to $2^{E_{\min}}$
◇) ***subrealmin*** (smallest positive subnormal number), equal to $\varepsilon_M\lambda$

Table I summarizes the values of the various constants encountered for the *tiny*, *single*, and *double* precision floating-point formats.

The error model presented in Equation (10) and Equation (11) is pessimistic by nature as it only gives an upper bound for the error. In some instances, it is possible to refine it to more accurate bounds. For example, we have for any finite floating-point number φ:

$$\mathsf{rndnr}\left\langle\frac{\varphi}{2}\right\rangle = \frac{\varphi}{2} + \eta \quad \text{with } \eta \in \left\{-\frac{\mu}{2}, 0, \frac{\mu}{2}\right\}. \tag{12}$$

The same holds when rounding toward zero; for upward and downward rounding, the possible values for η are reduced to, respectively, $\{0, \mu/2\}$ and $\{-\mu/2, 0\}$.

The proof of Equation (12) is easy: in the absence of underflow, the division by 2 is error-free since it amounts to decrementing the biased exponent by one; in case of underflow, the division by 2 is realized by shifting the fractional part one bit to the right (see Figure 4), which can only introduce an error of $\pm 2^{-(l(f)+1)} \times 2^{E_{\min}} = \pm\frac{\varepsilon_M}{2} \times 2^{E_{\min}} = \pm\frac{\mu}{2}$ if the discarded bit is a "1" and no error if the bit is a "0".

We may also enjoy an error-free addition/subtraction for certain operands, as stated by the following three theorems, which will be used in the next section. Beforehand, note that we may consider for all practical purposes that, for φ_1, φ_2 and φ_3 some floating-point numbers, $\mathsf{fl}\langle\varphi_1 + \varphi_2\rangle = \mathsf{fl}\langle\varphi_1 - (-\varphi_2)\rangle$ and $\mathsf{fl}\langle\varphi_1 - \varphi_2\rangle = \mathsf{fl}\langle\varphi_1 + (-\varphi_2)\rangle$ since taking the opposite of a floating-point number only involves flipping its sign bit, which is error-free.

THEOREM 2.1 (HAUSER'S THEOREM 3.4.1 [HAUSER 1996]). *If φ_1 and φ_2 are floating-point numbers, and if $\mathsf{fl}\langle\varphi_1 + \varphi_2\rangle$ underflows, then $\mathsf{fl}\langle\varphi_1 + \varphi_2\rangle = \varphi_1 + \varphi_2$.*

In effect, Theorem 2.1 means that η is always equal to 0 in Equation (11) when "$\top$" is an addition or a subtraction.

THEOREM 2.2 (HAUSER'S THEOREM 3.4.1A [HAUSER 1996]). *If φ_1 and φ_2 are floating-point numbers such that $|\varphi_1 + \varphi_2| < 2\lambda$, then $\mathsf{fl}\langle\varphi_1 + \varphi_2\rangle = \varphi_1 + \varphi_2$.*

THEOREM 2.3 (HAUSER'S THEOREM 3.4.2 [HAUSER 1996][10]). *If φ_1 and φ_2 are floating-point numbers such that $1/2 \leqslant \varphi_1/\varphi_2 \leqslant 2$, then $\mathsf{fl}\langle\varphi_1 - \varphi_2\rangle = \varphi_1 - \varphi_2$.*

Many other results to narrow the error bounds for some expressions may be found in the writings of Hauser [Hauser 1996], Goldberg [Goldberg 1991] or Higham [Higham 2002], to name a few, but the ones presented here will be enough for our purpose.

Hauser's theorems give sufficient conditions to have error-free additions and subtractions. On the other hand, these operations may generate large errors through *absorption* and *cancellation*.

Absorption. Consider the two *tiny* floating-point numbers $a = 1.000 \times 2^3$ and $b = 1.000 \times 2^{-1}$. To add them, we have first to align their exponents. This is done by scaling the operand with the smallest exponent:

$$\begin{array}{lrl} a & 1.000 & \times 2^3 \\ b & +0.000\mathit{1} & \times 2^3 \\ \hline & 1.000\mathit{1} & \times 2^3 \end{array}$$

The addition is performed with one extra bit for the significand of all values involved, the *guard bit*, which is necessary to ensure the validity of the error model of Equation (11). However, the result loses this guard bit when stored in the floating-point format on hand; if we round to the nearest-even, we then obtain $a + b = 1.000 \times 2^3 = a$. The *absolute error* is:

$$|\mathsf{rndnr}\langle a + b\rangle - (a + b)| = 2^{-1},$$

while the *relative error* is:

$$\frac{|\mathsf{rndnr}\langle a + b\rangle - (a + b)|}{|a + b|} = \frac{2^{-1}}{2^3 + 2^{-1}} = \frac{1}{17}.$$

As expected by the error model in Equation (11), the relative error is bounded by $\varepsilon_M/2 = 2^{-4} = 1/16$.

A loss of significance occurs whenever the difference in magnitude between the two operands of the addition leads to a shifting of some significant bits of one of the operands to the right of the rightmost bit of its fractional part. *Absorption* will occur when *all* the significant bits of one of the operands are shifted out.

Cancellation. Consider two nearly equal *tiny* numbers $a = 1.001 \times 2^3$ and $b = 1.000 \times 2^3$ and suppose they are the rounded approximations of earlier computation, respectively $a' = 1.00011 \times 2^3$ and $b' = 1.00010 \times 2^3$. If we perform the subtraction $a - b$, we get:

$$\begin{array}{lrl} a & 1.001 & \times 2^3 \\ b & -1.000 & \times 2^3 \\ \hline & 0.001 & \times 2^3 \end{array}$$

Even though the subtraction performed is error-free, we get the result "1" while $a' - b' = 2^{-2}$. All correct figures from a and b cancel each other out and only the wrong figures remain and are magnified in the process—the more so for large a and b. As a rule of thumb, we should therefore avoid subtracting quantities spoiled by previous errors.

[10] The better known Sterbenz's theorem [Sterbenz 1974] states the same result in a restricted form where underflows are forbidden [Higham 2002].

3. THE FLOATING-POINT MIDPOINT OPERATOR

As said at the beginning of Section 1, a naive floating-point implementation of Equation (1) must consider the rounding of $\rho = a + b$, for a and b two floating-point numbers[11]. A floating-point implementation of Equation (1) may fatally fail in two different ways with respect to the definition given by Equation (4):

Pitfall 1. For the interval $[-\infty, +\infty]$, $a + b$ returns a *not-a-number* (NaN), as defined by the IEEE 754 standard for the addition involving two infinities of differing signs. Mathematically, this does not violate the definition of midpoint, since the midpoint of $[-\infty, +\infty]$ is undefined. From a practical point of view, however, it violates Property (2) since a NaN would not be considered to be between $-\infty$ and $+\infty$. As said previously, for many current applications, it is better to return a finite value. Otherwise, the user must take proper care to check for the occurrence of a NaN each time some midpoint is computed, even for non-empty intervals;
Pitfall 2. If a and b have the same sign and their sum overflows, $a+b$ will be an infinity, which divided by two will still give an infinity[12].

We have presented in Section 1 the definition chosen for the midpoint operator. In the rest of this paper, we then require the implementation of midpoint to comply with Equation (2) and Equation (3), and to also have the properties:

$$\mathsf{m}(\boldsymbol{I}) \text{ is NaN} \iff \boldsymbol{I} = \varnothing.$$

That is, the midpoint operator should return a NaN only for empty intervals. And:

$$\forall \boldsymbol{I} = [a, b]: \quad a = -b \implies \mathsf{m}(\boldsymbol{I}) = 0.$$

That is, the midpoint of a *symmetric interval* should be 0 (the interval $[-\infty, +\infty]$ being considered symmetric).

The most straightforward implementation of the midpoint operator—such as used in, e.g., FI_LIB [Hofschuster and Krämer 1998]—has only one equation rounded to the nearest:

$$\mathsf{m}([a, b]) = \mathsf{rndnr}\left\langle \frac{a+b}{2} \right\rangle. \tag{13}$$

Evidently, it falls into both pitfalls highlighted above, and will return an infinite value in case $a + b$ overflows, and a NaN for the interval $[-\infty, +\infty]$.

The C++ Boost Interval Arithmetic Library [Brönnimann et al. 2006] uses a slightly more complex implementation of the midpoint operator, *viz.*:

$$\mathsf{m}([a, b]) = \begin{cases} -\texttt{realmax} & \text{if } a = -\infty; \\ \texttt{realmax} & \text{if } b = +\infty; \\ \mathsf{rndnr}\left\langle \frac{a+b}{2} \right\rangle & \text{otherwise.} \end{cases} \tag{Boost rev84}$$

The cases are not mutually exclusive; however, only one rule is used (the first one that applies, in the order given), noting that the actual implementation adds a zeroth case for empty interval inputs, for which it returns a NaN.

The error analysis for the third rule of Equation (Boost rev84)—also valid, obviously, for FI_LIB—is fairly simple with the model of Equation (11):

$$\mathsf{rndnr}\left\langle \frac{a+b}{2} \right\rangle = \frac{(a+b)(1+\delta_1)+\eta_1}{2}(1+\delta_2)+\eta_2.$$

[11] In the rest of this paper, we consider the double precision format, even though all results are applicable to other binary formats defined by the IEEE 754 standard [IEEE 1985] as well.
[12] See below for a *proviso* on this, though.

Thanks to Theorem 2.1, we have $\eta_1 = 0$. We also have $\delta_2 = 0$, since a division by two is error-free in the absence of underflow. Hence the simplified expression:

$$\mathsf{rndnr}\left\langle \frac{a+b}{2} \right\rangle = \frac{(a+b)}{2}(1+\delta_1) + \eta_2, \tag{14}$$

with $|\delta_1| \leqslant \varepsilon_M/2$ and $\eta_2 \in \{-\frac{\mu}{2}, 0, \frac{\mu}{2}\}$ (see Equation (12)). In the absence of overflow/underflow, we then have a correctly rounded result:

$$\mathsf{rndnr}\left\langle \frac{a+b}{2} \right\rangle = \frac{a+b}{2}(1+\delta_1),$$

which is the best we can expect.

Thanks to the first rule, the Boost library will not return a NaN for $\mathsf{m}([-\infty, +\infty])$ but $-$`realmax`. This result does not violate Equation (2), and Equation (3) is not relevant in the context of intervals with an infinite radius. It might, however, appear odd to the unsuspecting user to discover that the midpoint of a non-empty symmetric interval may not be 0.

That the Boost library may, in some instances, not fall prey to Pitfall 2 may come as a surprise; in fact, the occurrence of overflows depends on minute details of the actual compilation of the expression $\mathsf{rndnr}\langle(a+b)/2\rangle$: for example, on a computer with an ix86-based processor offering at least the SSE2 instruction set [Intel 2007, Chap. 11], there are essentially two ways to compute with double precision floating-point numbers:

(1) With the FPU instruction set and 80-bits registers stack [Intel 2007, Chap. 8]: computation is performed internally with an extended precision format having 64 bits of significand and 15 bits of exponent. It is still possible to restrict the size of the significand to 53 bits in order to emulate double precision[13] but this feature is rendered almost useless by the fact that it is not possible to restrict the size of the exponent;
(2) With the SSE2 instruction set and the SSE2 registers: computation is performed exclusively in double precision.

If a program that uses the Boost library is compiled without any particular option on an ix86-based processor, the expression $\mathsf{rndnr}\langle(a+b)/2\rangle$ might be computed as follows:

(1) The double precision variables a and b will be loaded into the FPU floating-point registers and promoted to the 80 bits extended format;
(2) The expression $\mathsf{rndnr}\langle(a+b)/2\rangle$ will be computed entirely with extended precision in the FPU registers, possibly incurring a first rounding in the extended format;
(3) The result of the expression will be demoted back into a double precision variable, possibly incurring a second rounding in the double format.

In this situation, the overflow cannot happen since $\mathsf{rndnr}\langle a+b\rangle$ is always representable by a finite number in extended precision; once divided by 2, the result is again representable in double precision. Additionnally, the two consecutive roundings may invalidate our error analysis since, even in the absence of underflow and overflow, the resulting error may then get larger than half a unit in the last place in rounding to nearest.

Now, suppose the program is compiled for an ix86 processor with a compiler option that instructs it to emit code to store back floating-point values into double precision variables after each operation[14] (this behavior has an obvious adverse effect on performance. It is useful, however, to promote repeatability of the computation across different platforms,

[13]Since version 4.3.0, the GNU C Compiler gcc offers the option `-mpc64` to do that easily.
[14]This is the `-ffloat-store` option of gcc, or the `-mp` option for the Intel C++ compiler.

some of which may not offer extended precision registers of the same size, if at all)[15]. In that case, the expression $\mathsf{rndnr}\langle \frac{a+b}{2} \rangle$ will be computed as follows:

(1) The double precision variables a and b will be loaded into the FPU floating-point registers and promoted to the 80 bits extended format;
(2) The expression $\mathsf{rndnr}\langle a+b \rangle$ will be computed with extended precision in the FPU registers;
(3) The value of $\mathsf{rndnr}\langle a+b \rangle$ will be stored back as a double precision floating-point number, at which point an overflow may occur;
(4) The value of $\mathsf{rndnr}\langle a+b \rangle$ will be loaded into an extended register, promoted to 80 bits, and then divided by 2;
(5) The result of the division will be stored in a double precision variable with some possible rounding.

In this situation, the Boost implementation of the midpoint operator may fail to comply with its requirements, and will return an infinite value as the midpoint of any interval with finite bounds whose absolute value of the sum is greater than `realmax`. The same problem would arise on architectures that do not rely internally on extended precision floating-point registers.

A different way to achieve the same failure on an ix86 processor relies on the possibility for modern compilers to transparently—or upon explicit request by the programmer—use the SIMD facilities offered by the SSE2 instruction set: some floating-point expressions may be computed in double precision SSE2 registers instead of the FPU extended precision registers. Once again, $\mathsf{rndnr}\langle a+b \rangle$ may overflow, as it will be computed in double precision only.[16] Such possibilities for unexpected failures are compounded by the advent of processors featuring new floating-point operations—such as *fused multiply-add* (fma), defined by the most recent version of the IEEE 754 standard for floating-point arithmetic [IEEE 2008]—whose rounding characteristics may invalidate the error analysis performed in this paper.

To avoid the overflow problem that affects the Boost implementation, Intlab V5.5 [Rump 1999b] uses the following formula:

$$\mathsf{m}([a,b]) = \begin{cases} 0 & \text{if } a = -\infty \text{ and } b = +\infty; \\ \mathsf{rndup}\langle \frac{a}{2} + \frac{b}{2} \rangle & \text{otherwise.} \end{cases} \qquad \text{(Intlab V5.5)}$$

The first rule protects explicitly against the creation of a NaN for the interval $[-\infty, +\infty]$, while the second rule avoids an overflow by first dividing the operands by two before adding the results, instead of the other way round.

The error analysis is as follows:

$$\mathsf{rndup}\left\langle \frac{a}{2} + \frac{b}{2} \right\rangle = \left[\left(\frac{a}{2}(1+\delta_1) + \eta_1 \right) + \left(\frac{b}{2}(1+\delta_2) + \eta_2 \right) \right] (1+\delta_3) + \eta_3.$$

Using Theorem 2.1 and Equation (12), we may simplify the expression as:

$$\mathsf{rndup}\left\langle \frac{a}{2} + \frac{b}{2} \right\rangle = \left(\frac{a+b}{2} + \eta_1 + \eta_2 \right) (1+\delta_3), \qquad (15)$$

[15] However, due to the unavoidably larger range of the exponent, perfect repeatability cannot be guaranteed for all cases.

[16] This transparent use of SIMD registers instead of FPU ones leads to another problem: the assembler instructions that set the rounding direction for the FPU and SSE2 instructions are separate; therefore, the rounding direction set by the programmer for some FPU computation will not apply if, unbeknownst to him, the compiler chooses to use SSE2 instructions instead. However, this problem disappears if the programmer can make use of recent enough versions of high level functions such as the C99 `fesetround()` and Microsoft's `_controlfp()`, which set the rounding direction for both FPU and SSE2 intructions.

with $\eta_1 \in \{0, \mu/2\}$, $\eta_2 \in \{0, \mu/2\}$, and $0 \leqslant \delta_3 < \varepsilon_M$, since results are rounded upward now. In the absence of underflow/overflow, Equation (15) simplifies to:

$$\mathsf{rndup}\left\langle \frac{a}{2} + \frac{b}{2} \right\rangle = \frac{a+b}{2}(1+\delta_3),$$

meaning that the result is correctly rounded (though not to the nearest). Note that, in practice, the quality of this formula is not as good as Boost's since the bound on δ_3 in Equation (15) is twice the one on δ_1 in Equation (14).

With this algorithm, the midpoint of an interval with one infinite bound (but not both) is this infinity (for example: $\mathsf{m}([-\infty, 3]) = -\infty$ and $\mathsf{m}([3, +\infty]) = +\infty$). In addition, a symmetric interval with finite bounds may have a midpoint different from 0: the midpoint of the interval $[-\mu, \mu]$ is computed as μ because the expressions $\mathsf{rndup}\langle 0.5 \times -\mu \rangle$ and $\mathsf{rndup}\langle 0.5 \times \mu \rangle$ are rounded, respectively, to 0 and μ. For the interval $[-\mu, \mu]$, this has the unfortunate effect of violating Equation (3) since 0 is a representable floating-point number closer to the midpoint of the interval.

Additionally and more importantly, Intlab V5.5 may also violate Equation (2). Consider the reduction of Equation (15) for subnormal inputs:

$$\mathsf{rndup}\left\langle \frac{a}{2} + \frac{b}{2} \right\rangle = \frac{a+b}{2} + \eta_1 + \eta_2.$$

We have $\eta_1 + \eta_2 \in \{0, \mu/2, \mu\}$, which means that in some cases, the result may not be correctly rounded. Take for example, the interval $[\mu, \mu]$. Since we have:

$$\mathsf{rndup}\langle 0.5 \times \mu \rangle = \mu,$$

it follows:

$$\mathsf{m}([\mu, \mu]) = 2\mu \notin [\mu, \mu].$$

BIAS 2.0.8 [Knüppel 1994] follows a different path and computes the midpoint as follows:

$$\mathsf{m}([a, b]) = \mathsf{rndup}\left\langle a + \frac{b-a}{2} \right\rangle. \tag{BIAS 2.0.8}$$

Despite using a subtraction in place of an addition, this algorithm has the same flaws as a naive implementation of midpoint: if $b - a$ overflows (say, because a and b are huge, with differing signs), the result will be infinite; additionally, $\mathsf{m}([-\infty, +\infty])$ is a NaN because $b - a = +\infty - -\infty = +\infty$ and $a + (b - a) = -\infty + \infty = \text{NaN}$.

The error analysis is also interesting:

$$\mathsf{rndup}\left\langle a + \frac{b-a}{2} \right\rangle = \left[a + \frac{(b-a)(1+\delta_1) + \eta_1}{2}(1+\delta_2) + \eta_2 \right](1+\delta_3) + \eta_3,$$

which reduces to:

$$\mathsf{rndup}\left\langle a + \frac{b-a}{2} \right\rangle = \left(a + \frac{(b-a)}{2}(1+\delta_1) + \eta_2 \right)(1+\delta_3), \tag{16}$$

with $\eta_2 \in \{0, \mu/2\}$, $0 \leqslant \delta_1 < \varepsilon_M$ and $0 \leqslant \delta_3 < \varepsilon_M$. In the absence of overflow/underflow, we get:

$$\mathsf{rndup}\left\langle a + \frac{b-a}{2} \right\rangle = \frac{a+b}{2} + \left(\delta_1 \frac{b-a}{2} + \delta_3 \frac{a+b}{2} + \delta_1 \delta_3 \frac{b-a}{2} \right).$$

In theory, BIAS's formula is then even worse than Intlab V5.5's in the absence of underflow. On the other hand, the reduction of Equation (16) for the case of subnormal inputs shows that it is not affected by Intlab V5.5's problem:

$$\mathsf{rndup}\left\langle a + \frac{b-a}{2} \right\rangle = \frac{a+b}{2} + \eta_2.$$

For subnormal inputs, BIAS computes a correctly rounded result.

CXSC 2.4.0 [Hammer et al. 1997] offers two functions to compute the midpoint: a "fast" one, `Mid(I)`, and an accurate one, `mid(I)`. The fast one computes the midpoint by a formula akin to both BIAS's and Intlab's, except for the rounding, which is now downward:

$$\mathsf{m}([a,b]) = \mathsf{rnddn}\left\langle a + \left(\frac{b}{2} - \frac{a}{2}\right) \right\rangle. \qquad \text{(CXSC 2.4.0)}$$

That formula is still subject to the NaN problem whenever $[a,b]=[-\infty,+\infty]$. The accurate version uses extended precision, and then, contrary to the fast version, it always returns the correctly rounded value for the midpoint, except for the interval $[-\infty,+\infty]$. For that interval, CXSC aborts abruptly with an exception.

The error analysis for the "fast" midpoint operator of CXSC gives:

$$\mathsf{rnddn}\left\langle a + \left(\frac{b}{2} - \frac{a}{2}\right) \right\rangle = \left[a + \left(\left(\frac{b}{2}(1+\delta_1)+\eta_1\right) - \left(\frac{a}{2}(1+\delta_2)+\eta_2\right)\right)(1+\delta_3) + \eta_3\right](1+\delta_4) + \eta_4,$$

which simplifies to:

$$\mathsf{rnddn}\left\langle a + (\frac{b}{2} - \frac{a}{2}) \right\rangle = \left[a + \left(\frac{b-a}{2} + \eta_1 - \eta_2\right)(1+\delta_3)\right](1+\delta_4), \qquad (17)$$

with $\eta_1 \in \{-\mu/2, 0\}$, $\eta_2 \in \{-\mu/2, 0\}$, $-\varepsilon_M < \delta_3 \leqslant 0$, and $-\varepsilon_M < \delta_4 \leqslant 0$.

In the absence of overflow/underflow, CXSC's formula is as bad as BIAS's:

$$\mathsf{rnddn}\left\langle a + (\frac{b}{2} - \frac{a}{2}) \right\rangle = \frac{a+b}{2} + \left(\delta_3 \frac{b-a}{2} + \delta_4 \frac{a+b}{2} + \delta_3\delta_4 \frac{b-a}{2}\right).$$

For subnormal inputs, we have:

$$\mathsf{rnddn}\left\langle a + (\frac{b}{2} - \frac{a}{2}) \right\rangle = \frac{a+b}{2} + \eta_1 - \eta_2,$$

with $\eta_1 - \eta_2 \in \{-\mu/2, 0, \mu/2\}$.

Filib++ 2.0 [Lerch et al. 2006] uses the same formula as Intlab V5.5 rounded differently, with additional rules to avoid some of the pitfalls previously encountered:

$$\mathsf{m}([a,b]) = \begin{cases} a & \text{if } a = b, \\ 0 & \text{if } a = -b, \\ \mathsf{rndnr}\langle \frac{a}{2} + \frac{b}{2} \rangle & \text{otherwise.} \end{cases} \qquad \text{(Filib++ 2.0)}$$

The error analysis for Filib++'s formula leads to the same expression as for Intlab V5.5, except for the bounds on the errors:

$$\mathsf{rndnr}\left\langle \frac{a}{2} + \frac{b}{2} \right\rangle = \left(\frac{a+b}{2} + \eta_1 + \eta_2\right)(1+\delta_3), \qquad (18)$$

with $\eta_1 \in \{-\mu/2, 0, \mu/2\}$, $\eta_2 \in \{-\mu/2, 0, \mu/2\}$, and $|\delta_3| \leqslant \varepsilon_M/2$.

As for Intlab V5.5, the result is correctly rounded whenever no overflow/underflow occurs. On the other hand, for subnormal inputs, we still get:

$$\mathsf{rndnr}\left\langle \frac{a}{2} + \frac{b}{2} \right\rangle = \frac{a+b}{2} + \eta_1 + \eta_2,$$

with $\eta_1 + \eta_2 \in \{-\mu, -\mu/2, 0, \mu/2, \mu\}$, which means that the computed midpoint may be one of the floating-point numbers that surround the correctly rounded midpoint. For a degenerate interval reduced to one point, the first rule protects against computing a midpoint outside the interval, as was the case with Intlab V.5.5; obviously, the formula ensures that the midpoint of an interval with a width greater or equal to 2μ is included in it. For an interval whose bounds a and b are consecutive subnormal floating-point numbers (hence, with a width precisely equal to μ), it suffices to notice that exactly one of them has a rightmost bit of the fractional part equal to "1" and the other one has the corresponding bit equal to "0". Consequently, $\eta_1 + \eta_2 \in \{-\mu/2, 0, \mu/2\}$ and the inclusion is, once again, ensured.

The second rule in Equation (Filib++ 2.0) is more general than the first one of Intlab V5.5: it protects against the computation of a NaN for the interval $[-\infty, +\infty]$ as well as ensuring that symmetric intervals in general have 0 as their midpoint (another "flaw" of the Intlab V5.5 formula). The expression in the third rule cannot overflow for finite a and b, which makes the whole formula immune to the overflow problem. On the other hand, the midpoint of any interval with an infinite bound (but not both) is infinite (*e.g.*: $\mathsf{m}([3, +\infty]) = +\infty$). This result does not violate Equation (2) and Equation (3); it may nevertheless lead to serious trouble when the returned value is used in, say, a bisection algorithm or for a centered form without proper care.

The recent Version 6 of the Intlab package has a different algorithm from Version 5.5 to compute the midpoint of an interval:

$$\mathsf{m}([a,b]) = \begin{cases} 0 & \text{if } a = -\infty \text{ and } b = +\infty, \\ b & \text{if } a = -\infty, \\ a & \text{if } b = +\infty, \\ \mathsf{rndnr}\langle a + (\frac{b}{2} - \frac{a}{2}) \rangle & \text{otherwise.} \end{cases} \qquad \text{(Intlab V6)}$$

Once again, these rules are not mutually exclusive, but the first that applies is executed to the exclusion of the others. The first rule explicitly avoids the NaN problem for the interval $[-\infty, +\infty]$; the second and third rules ensure that the midpoint of an interval with one infinite bound (and one only) is finite and equal to the other—finite—bound. This is one possible choice among many to resolve the problem, though different from Equation (4).

Apart from the rounding direction, the formula in the fourth rule is the same as CXSC's formula, and consequently, an error analysis along the same lines may be made:

$$\mathsf{rndnr}\left\langle a + (\frac{b}{2} - \frac{a}{2}) \right\rangle = \left[a + \left((\frac{b}{2} + \eta_1) - (\frac{a}{2} + \eta_2) \right) (1 + \delta_3) \right] (1 + \delta_4), \qquad (19)$$

with $\eta_1 \in \{-\mu/2, 0, \mu/2\}$, $\eta_2 \in \{-\mu/2, 0, \mu/2\}$, $|\delta_3| \leqslant \varepsilon_M/2$, and $|\delta_4| \leqslant \varepsilon_M/2$.

For subnormal inputs, it may violate Equation (3), as evident from the error formula:

$$\mathsf{rndnr}\left\langle a + (\frac{b}{2} - \frac{a}{2}) \right\rangle = \frac{a+b}{2} + \eta_1 - \eta_2,$$

with $\eta_1 - \eta_2 \in \{-\mu, -\mu/2, 0, \mu/2, \mu\}$. For example, with Intlab V6's formula, we have:

$$\mathsf{m}([-\mu, \mu]) = -\mu.$$

On the other hand, for finite a and b, the fourth rule is immune to overflow, and it ensures that Equation (2) is satisfied, as proven by Prop. 3.1.

PROPOSITION 3.1. *Given a and b two finite floating-point numbers, with $a \leqslant b$, we have:*

$$a \leqslant \mathsf{rndnr}\left\langle a + \left(\frac{b}{2} - \frac{a}{2}\right)\right\rangle \leqslant b.$$

PROOF. *Since $a \leqslant b$, we have $\mathsf{rndnr}\langle a/2\rangle \leqslant \mathsf{rndnr}\langle b/2\rangle$, thanks to rounding monotonicity. This is true even if an underflow occurs. As a consequence, $\mathsf{rndnr}\langle b/2 - a/2\rangle \geqslant 0$, and then $\mathsf{rndnr}\langle a + (b/2 - a/2)\rangle \geqslant a$, again by monotonicity of rounding. For finite a and b, $\mathsf{rndnr}\langle b/2 - a/2\rangle$ cannot overflow.*

Let us now prove that $\mathsf{rndnr}\langle a + (b/2 - a/2)\rangle \leqslant b$. We first prove that $\mathsf{rndnr}\langle b/2 - a/2\rangle \leqslant b - a$, restricting the discussion to the case $a \neq b$ since the proof is trivial otherwise. We have:

$$\mathsf{rndnr}\left\langle \frac{b}{2} - \frac{a}{2}\right\rangle = \left[\left(\frac{b}{2} + \eta_1\right) - \left(\frac{a}{2} + \eta_2\right)\right](1 + \delta_3)\,. \tag{20}$$

Using Equation (20), let us now find the conditions such that $\mathsf{rndnr}\langle b/2 - a/2\rangle > b - a$:

$$\begin{aligned}\mathsf{rndnr}\left\langle \frac{b}{2} - \frac{a}{2}\right\rangle > b - a &\iff (b-a)\frac{1+\delta_3}{2} + (\eta_1 - \eta_2)(1+\delta_3) > b - a\\ &\iff b - a < 2(\eta_1 - \eta_2)\frac{1+\delta_3}{1-\delta_3}.\end{aligned}$$

We have $|\delta_3| \leqslant \varepsilon_M/2$, $|\eta_1| \leqslant \mu/2$, and $|\eta_2| \leqslant \mu/2$. As soon as $l(f) > 1$, $(1+\delta_3)/(1-\delta_3)$ is strictly smaller than 2. Hence, $b-a$ has to be less than 4μ for $\mathsf{rndnr}\langle b/2 - a/2\rangle$ to be greater than $b - a$.

We then deduce that $\mathsf{rndnr}\langle b/2 - a/2\rangle$ is safely strictly less than λ for any floating-point format of practical use (just use $b - a < 4\mu$ in Equation (20)), that is $\mathsf{rndnr}\langle b/2 - a/2\rangle$ underflows. In that case, $\delta_3 = 0$, thanks again to Theorem 2.1, and then we have:

$$\begin{aligned}\mathsf{rndnr}\left\langle \frac{b}{2} - \frac{a}{2}\right\rangle > b - a &\iff \frac{b-a}{2} + \eta_1 - \eta_2 > b - a\\ &\iff b - a < 2(\eta_1 - \eta_2).\end{aligned}$$

With the known bounds on η_1 and η_2, we now deduce that $b-a$ must be strictly less than 2μ for $\mathsf{rndnr}\langle b/2 - a/2\rangle$ to be greater than $b-a$. But $b-a$ must be a strictly positive floating-point number (we have $b > a$, and $b-a < \lambda$ implies that Theorem 2.1 applies). Consequently, $b-a$ must be equal to μ, which means that b and a are two consecutive floating-point numbers. As a result, either a or b has an "even" binary significand (rightmost bit equal to 0), which means that either $\eta_1 = 0$ or $\eta_2 = 0$. We may then refine our condition:

$$\begin{aligned}\mathsf{rndnr}\left\langle \frac{b}{2} - \frac{a}{2}\right\rangle > b - a &\iff \frac{b-a}{2} + \eta > b - a \quad \text{with } |\eta| \leqslant \mu/2\\ &\iff b - a < \mu.\end{aligned}$$

This condition is not possible for $a \neq b$ since there is no positive floating-point number smaller than μ. Consequently:

$$\mathsf{rndnr}\left\langle \frac{b}{2} - \frac{a}{2}\right\rangle \leqslant b - a \qquad \forall (a,b) \in \mathbb{F}^2.$$

We then have:

$$a + \mathsf{rndnr}\left\langle \frac{b}{2} - \frac{a}{2}\right\rangle \leqslant b.$$

And, by monotonicity of rounding:

$$\mathsf{rndnr}\left\langle a + (\frac{b}{2} - \frac{a}{2}) \right\rangle \leqslant b.$$

□

The interval arithmetic library Gaol 4.0RC [Goualard 2010] implements the midpoint operator by taking advantage of Hauser's theorems:

$$\mathsf{m}([a,b]) = \begin{cases} 0 & \text{if } a = -b, \\ -\texttt{realmax} & \text{if } a = -\infty, \\ \texttt{realmax} & \text{if } b = +\infty, \\ \mathsf{rndnr}\langle (a - \frac{a}{2}) + \frac{b}{2} \rangle & \text{otherwise.} \end{cases} \qquad \text{(Gaol 4.0RC)}$$

As before, only the first rule whose condition is true is considered. Case 4 rewrites the expression "$a + (b/2 - a/2)$" from Intlab V6 as "$(a - a/2) + b/2$", a small algebraic manipulation indeed, but one that can have a significant impact on the accuracy of the whole expression. In effect, the error analysis gives:

$$\mathsf{rndnr}\left\langle \left(a - \frac{a}{2}\right) + \frac{b}{2} \right\rangle = \left[\left(a - \left(\frac{a}{2}(1+\delta_1) + \eta_1\right)\right)(1+\delta_2) + \eta_2 + \left(\frac{b}{2}(1+\delta_3) + \eta_3\right) \right](1+\delta_4) + \eta_4. \quad (21)$$

Lemma 3.2. *For any finite floating-point number $a \in \mathbb{F}$, we have:*

$$\mathsf{rndnr}\left\langle a - \frac{a}{2} \right\rangle = a - \left(\frac{a}{2} + \eta\right), \quad \text{with } \eta \in \left\{-\frac{\mu}{2}, 0, \frac{\mu}{2}\right\}.$$

Proof. *There are two cases to consider: let us first suppose no underflow occurs when halving a. Then, we have $\mathsf{rndnr}\langle a/2 \rangle = a/2$ since halving is error-free in the absence of underflow. We may then use Hauser's Theorem 2.3 to state that $\mathsf{rndnr}\langle a - a/2 \rangle = a - a/2$.*

Now, suppose an underflow occurs when halving a; then, $\mathsf{rndnr}\langle a/2 \rangle = a/2 + \eta$. This event is only possible if $a/2$ is strictly smaller than λ (by definition), and this can happen only if a is strictly smaller than 2λ. But then, $\mathsf{rndnr}\langle a - (a/2 + \eta) \rangle$ is also strictly smaller than 2λ, and we may then use Theorem 2.2 to state that $\mathsf{rndnr}\langle a - a/2 \rangle = a - (a/2 + \eta)$. □

Using Lemma 3.2 and the property on halving given by Equation (12), we are able to simplify Equation (21) to:

$$\mathsf{rndnr}\left\langle \left(a - \frac{a}{2}\right) + \frac{b}{2} \right\rangle = \left(\frac{a+b}{2} + \eta_3 - \eta_1\right)(1+\delta_4), \quad (22)$$

with $\eta_1 \in \{-\mu/2, 0, \mu/2\}$, $\eta_3 \in \{-\mu/2, 0, \mu/2\}$, $|\delta_4| \leqslant \varepsilon_M/2$. It is then straightforward to show that containment is always ensured: no overflow can occur in computing $(a - a/2) + b/2$ and, in the absence of underflow, a correctly rounded midpoint is computed; on the other hand, in case of underflow, the error is bounded by μ and not $\mu/2$. The benefit of using "$a - a/2$" instead of "$a/2$" directly is apparent when we compare Equation (22) with Equation (18): when $a = b$, we have $\eta_1 + \eta_2 = \pm\mu$ in the worst case for Equation (18), and $\eta_3 - \eta_1 = 0$ in all cases for Equation (22) (errors in halving a and b cancel out when $a = b$).

The formulae investigated so far all have defects with respect to the definition in Equation (4). Table II synthesizes the error formulae in three cases:

— **GC**. The general case, where an underflow may or may not occur at any step of the computation (for the experiments, overflows are allowed in the general case as well, although the formulae in Table II are then no longer relevant);
— **No OF/SN**. The case where no overflow nor any underflow occurs at any step of the computation, and neither infinities nor subnormal numbers are used as bounds of the test intervals;
— **SN**. The case where all quantities manipulated and produced (counting the initial inputs) are subnormal numbers.

Table II. Synthesis of worst-case error bounds.

Method	Error GC	No OF/SN	SN
Boost	$\frac{a+b}{2}(1+\delta_1)+\eta_2$	$\frac{a+b}{2}(1+\delta_1)$	$\frac{a+b}{2}+\eta_2$
$\mathsf{rndnr}\langle(a+b)/2\rangle$	$\lvert\eta_2\rvert \in \{0,\mu/2\}, \lvert\delta_1\rvert \leqslant \varepsilon_M/2$		
Intlab V5.5	$\left((\frac{a}{2}+\eta_1)+(\frac{b}{2}+\eta_2)\right)(1+\delta_3)$	$\frac{a+b}{2}(1+\delta_3)$	$\frac{a+b}{2}+\eta_1+\eta_2$
$\mathsf{rndup}\langle a/2+b/2\rangle$	$\eta_1 \in \{0,\mu/2\}, \eta_2 \in \{0,\mu/2\}, 0 \leqslant \delta_3 < \varepsilon_M$		
BIAS	$\left(a+\frac{(b-a)}{2}(1+\delta_1)+\eta_2\right)(1+\delta_3)$	$\left(a+\frac{(b-a)}{2}(1+\delta_1)\right)(1+\delta_3)$	$\frac{a+b}{2}+\eta_2$
$\mathsf{rndup}\langle a+(b-a)/2\rangle$	$\eta_2 \in \{0,\mu/2\}, 0 \leqslant \delta_1 < \varepsilon_M, 0 \leqslant \delta_3 < \varepsilon_M$		
CXSC	$\left[a+\left((\frac{b}{2}+\eta_1)-(\frac{a}{2}+\eta_2)\right)(1+\delta_3)\right](1+\delta_4)$	$\left[a+\frac{(b-a)}{2}(1+\delta_3)\right](1+\delta_4)$	$\frac{a+b}{2}+\eta_1-\eta_2$
$\mathsf{rnddn}\langle a+(b/2-a/2)\rangle$	$\eta_1 \in \{-\mu/2,0\}, \eta_2 \in \{-\mu/2,0\}, -\varepsilon_M < \delta_3 \leqslant 0, -\varepsilon_M < \delta_4 \leqslant 0$		
Filib++	$\left((\frac{a}{2}+\eta_1)+(\frac{b}{2}+\eta_2)\right)(1+\delta_3)$	$\frac{a+b}{2}(1+\delta_3)$	$\frac{a+b}{2}+\eta_1+\eta_2$
$\mathsf{rndnr}\langle a/2+b/2\rangle$	$\lvert\eta_1\rvert \in \{0,\mu/2\}, \lvert\eta_2\rvert \in \{0,\mu/2\}, \lvert\delta_3\rvert \leqslant \varepsilon_M/2$		
Intlab V6	$\left[a+\left((\frac{b}{2}+\eta_1)-(\frac{a}{2}+\eta_2)\right)(1+\delta_3)\right](1+\delta_4)$	$\left[a+\frac{(b-a)}{2}(1+\delta_3)\right](1+\delta_4)$	$\frac{a+b}{2}+\eta_1-\eta_2$
$\mathsf{rndnr}\langle a+(b/2-a/2)\rangle$	$\lvert\eta_1\rvert \in \{0,\mu/2\}, \lvert\eta_2\rvert \in \{0,\mu/2\}, \lvert\delta_3\rvert \leqslant \varepsilon_M/2, \lvert\delta_4\rvert \leqslant \varepsilon_M/2$		
Gaol 4.0RC	$(\frac{a+b}{2}+\eta_3-\eta_1)(1+\delta_4)$	$\frac{a+b}{2}(1+\delta_4)$	$\frac{a+b}{2}+\eta_3-\eta_1$
$\mathsf{rndnr}\langle(a-a/2)+b/2\rangle$	$\lvert\eta_1\rvert \in \{0,\mu/2\}, \lvert\eta_3\rvert \in \{0,\mu/2\}, \lvert\delta_4\rvert \leqslant \varepsilon_M/2$		

For the general case (GC), Table II presents worst-case error bounds that may not be reachable in practice. On close inspection, we can even refine these bounds. Consider for example the formula for Filib++:

$$\mathsf{rndnr}\langle a/2+b/2\rangle = \left((\frac{a}{2}+\eta_1)+(\frac{b}{2}+\eta_2)\right)(1+\delta_3).$$

It is not possible for η_1, η_2, and δ_3 to be simultaneously non-null: according to Theorem 2.2, for δ_3 to be different from 0, we must have:

$$\left|(\frac{a}{2}+\eta_1)+(\frac{b}{2}+\eta_2)\right| \geqslant 2\lambda. \tag{23}$$

But, for η_1 and η_2 to be both different from 0, we must have $|a| < 2\lambda$ and $|b| < 2\lambda$, and then, $|(\frac{a}{2}+\eta_1)|$ and $|(\frac{b}{2}+\eta_2)|$ are each at most equal to λ. As a consequence, the only case where Equation (23) is satisfied is when $|(\frac{a}{2}+\eta_1)+(\frac{b}{2}+\eta_2)|$ is precisely equal to 2λ, for which case $\delta_3 = 0$.

On the other hand, for the two other columns of the table, it is possible to construct examples for which all δs and ηs are non-null simultaneously, and the worst-case error bound is reached.

With respect to error bounds, the most promising expressions are:

— $\mathsf{rndnr}\langle(a+b)/2\rangle$ (Boost), which has the best error bounds for all situations, but may return an infinite value if $a+b$ overflows;
— $\mathsf{rndnr}\langle a/2+b/2\rangle$ (Filib++), which may return an infinite value for semi-unbounded intervals (unbounded intervals being treated separately);
— $\mathsf{rndnr}\langle(a-a/2)+b/2\rangle$ (Gaol 4.0RC).

The algorithms that use the first two expressions can be amended to remove the associated flaws:

— Boost's algorithm (Equation (Boost rev84)) can be supplemented by some *a posteriori* test to check whether the output overflowed; if it is the case, the midpoint is recomputed using a different formula for which an overflow cannot occur (e.g., $\mathsf{rndnr}\langle a/2+b/2\rangle$);
— Amending Filib++'s algorithm is easy by adding cases as follows:

$$\mathsf{m}([a,b]) = \begin{cases} a & \text{if } a = b, \\ 0 & \text{if } a = -b, \\ -\texttt{realmax} & \text{if } a = -\infty, \\ \texttt{realmax} & \text{if } b = +\infty, \\ \mathsf{rndnr}\langle \frac{a}{2} + \frac{b}{2} \rangle & \text{otherwise.} \end{cases} \tag{24}$$

From Theory to Practice

So far, we have only theoretically analyzed the various formulae to compute the midpoint of an interval $[a,b]$. According to Table II, we should be able to separate the various methods into four classes for the case where no underflow nor any overflow occurs:

— Boost, Filib++ and Gaol, which deliver a correctly rounded result to the *unit roundoff* (half the epsilon of the machine);
— Intlab V5.5, which delivers a correctly rounded result to the epsilon of the machine;
— Intlab V6, which delivers a result with an error proportional to the unit roundoff;
— Bias and CXSC, which deliver a result with an error that may get twice as large as Intlab V6's for the same inputs.

This is indeed verified by the experiments, as shown in Table III, where the first column "Disc" indicates the number of floating-point numbers between the correctly rounded midpoint and the midpoint computed by the method tested. The last row "Fail" in the table indicates the number of times a midpoint computed was not included in the interval input. The table reports the results for the computation of the midpoint of ten million non-empty random intervals.

The comparison of all formulae when subnormal numbers are produced for each operation shows less drastic differences (see Table IV): the best we can compute in that case is $(a+b)/2+\eta$, with $|\eta| \in \{0, \mu/2\}$, which is exactly what is computed by Boost. BIAS exhibits the same error bound, but the computed result may differ by one unit in the last place from Boost's, due to the upward rounding. Intlab V5.5 shows a discrepancy of one unit in the last place almost half of the time: the error is $\eta_1+\eta_2$ with η_1 and η_2 both positive or null, which means that they can never cancel each other out. On the other hand, CXSC offers a good accuracy because the error term is $\eta_1-\eta_2$ with both η_1 and η_2 being of the same sign. Lastly, Filib++, Intlab V6 and Gaol are of similar accuracy, with Intlab V6 and Gaol being rigorously equivalent, as Table II suggests already.

If we allow both underflows and overflows (see Table V), we now have five classes:

Table III. Comparison of the quality of the midpoint() implementations — No OF/SN.

Disc.	Boost	Intlab V5.5	BIAS	CXSC	Filib++	Intlab V6	Gaol 4.0RC
0	10000000	5008451	2508795	2508307	10000000	9864663	10000000
1		4991549	7486334	7486883		132983	
2			3132	3028		1178	
3			460	467			
4			585	569		603	
6			103	139			
8			287	291		294	
12			18	34			
16			126	125		127	
24			7	3			
32			73	74		74	
48			1	2			
64			36	36		36	
96			1				
128			25	25		25	
256			8	8		8	
512			4	4		4	
1024			4	4		4	
16384			1	1		1	
Fail	0	0	0	0	0	0	0

Table IV. Comparison of the quality of the midpoint() implementations — SN.

Disc.	Boost	Intlab V5.5	BIAS	CXSC	Filib++	Intlab V6	Gaol 4.0RC
0	10000000	5001983	7502568	7500491	6251593	6250468	6250468
1		4998017	2497432	2499509	3748407	3749532	3749532
Fail	0	0	0	0	0	0	0

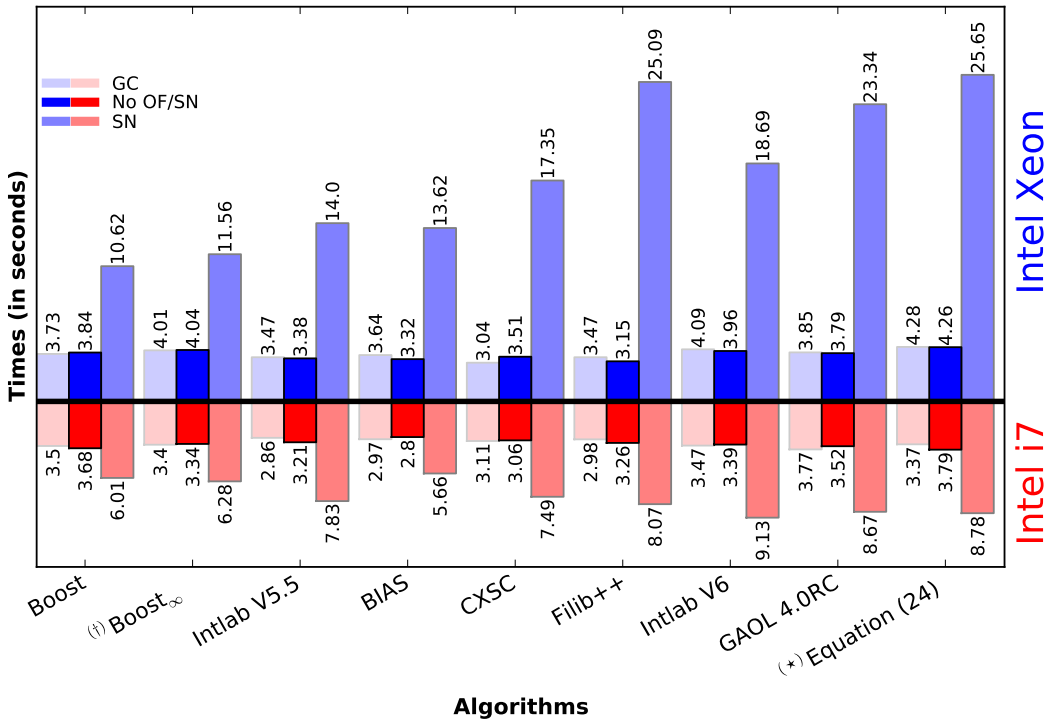
Table V. Comparison of the quality of the midpoint() implementations — GC.

Disc.	Boost	Intlab V5.5	BIAS	CXSC	Filib++	Intlab V6	H-method
0	9990382	5002986	2506554	2506186	9990379	9855400	9990381
1		4987404	7478969	7479251	11	132539	9
2			3104	3144		1226	
3			428	468			
4			601	612		632	
6			105	116			
8			277	275		283	
12			31	20			
16			160	161		162	
24			6	8			
32			68	70		70	
48			2	1			
64			46	46		46	
96			2				
128			15	15		15	
256			6	6		6	
512			5	5		5	
1024			3	3		3	
8192			2	2		2	
32768			1	1		1	
65536	8		5				
Fail	8	0	4847	4842	0	0	0

— Boost, which always returns a correctly rounded result, except when overflows make it fail to ensure inclusion of the midpoint;
— Gaol and Filib++, which never fail to meet the requirement of inclusion while never departing by more than one unit in the last place from the correctly rounded result;
— Intlab V5.5, which delivers half of the time a result departing from the correctly rounded result by one ulp. Although it is not apparent from Table V, recall that it may fail to ensure containment in some rare cases (see page 13);
— Intlab V6, whose results may significantly depart from the correctly rounded one, although it always meets the inclusion requirement;
— BIAS and CXSC, whose results may significantly depart from the correctly rounded one, and which may not meet the inclusion requirement.

Discrepancies for intervals with at least one infinite bound are not reported in Table V.

Figure 7 compares the various methods performance-wise for two different platforms (Intel Xeon E5520 on Ubuntu 10.04.4, and Intel i7 "*Sandy Bridge*" on Ubuntu 11.04). All the methods have been implemented in C++ in the same environment. Times are given in seconds for the computation of the midpoint of fifty million non-empty random intervals, with the same three categories (GC, No OF/SN, SN) as in Table II. The C++ compiler used is GCC 4.8.0 with the options "`-ffloat-store -O0 -mpc64 -frounding-math`".



(†) *Boost formula where Filib++'s formula is used in case of overflow*

(⋆) *Filib++ formula with tests added to avoid returning an infinite value.*

Fig. 7. Performances of nine `midpoint()` implementations for two CPU architectures.

Note the difference in time between the first two categories ("GC" and "No OF/SN") and the last one ("SN") for both architectures: subnormal numbers have to be handled as

exceptional values, which slows the computation down dramatically. However, Intel Xeon and Intel i7 "*Sandy Bridge*" differ in their handling of subnormal numbers, the latter implementing part of its support in hardware rather than as microcode exceptions [Fog 2012, p. 102], which drastically reduces the overhead incurred in using subnormals.

Table VI. Midpoint formulae shortcomings.

Package	NaN	Inf	NC	SYM	IR
Boost rev84 (P. 10)		✓		✓	
Boost_∞				✓	
Intlab V5.5 (P. 12)		✓	✓	✓	✓
BIAS 2.0.8 (P. 13)	✓	✓			✓
CXSC 2.4.0 (Mid) (P. 14)	✓	✓			✓
Filib++ 2.0 (P. 14)		✓			✓
Intlab V6 (P. 15)				✓	✓
Gaol 4.0RC (P. 17)				✓	✓
Equation (24) (P. 19)					✓

NaN. Returning a NaN for a non-empty interval (e.g., for $\mathsf{m}([-\infty, +\infty])$).
Inf. Returning an infinite midpoint (e.g., in case of overflow in the addition $a + b$, or if a bound is infinite).
NC. Failing to meet the containment requirement (e.g., $\mathsf{m}([a, b]) \notin [a, b]$ in case of underflow).
SYM. Violation of Equation (3): not returning 0 for a non-empty symmetric interval.
IR. Violation of Equation (3): returning a value different from the correctly rounded center of the interval in the absence of overflow.

Table VI summarizes the potential shortcomings of the midpoint implementation for the packages we have considered. Constraint (3) is violated by almost all packages. Pending further investigation, SYM and IR may be considered benign errors because they should not compromise the reliability of algorithms based on packages that do exhibit them. All other problems may put reliability in jeopardy for applications that expect a definition of the midpoint operator matching Equation (4).

The corrected versions of Boost's and Filib++'s formulae (Boost_∞ and Equation (24), respectively) are the algorithms with the fewest number of "defects". However, it seems easier to avoid the symmetry problem of Boost_∞ by returning 0 when the input is $[-\infty, +\infty]$ than to correct Equation (24)'s rounding problem in presence of subnormal numbers.

Performance-wise, Boost_∞ is also not an unreasonable choice if we keep in mind that slightly faster methods may lead to incorrect results. Experiments also show that adding a test for the interval $[-\infty, +\infty]$ as input to return 0 does not incur any significant overhead, and gives us an algorithm that matches the requirements of Equation (4). The resulting algorithm to compute the midpoint would then be the one given in pseudo-code in Table VII.

4. CONCLUSION

The IEEE Standards Association is currently supporting the definition of Standard P1788 [IEEE 1788 2013] for interval arithmetic. There is an ongoing effort to rigorously define the properties that functions such as the midpoint operator should exhibit. As the numerous exchanges on the IEEE P1788 mailing list have shown, the consensus on these properties is not trivial to reach, as different properties may be needed for different applications, which might give rise eventually to several specialized midpoint operators[17]. Motion 37, passed in November 2012, defined after some heated debate the midpoint operator as stated in Equation (4).

[17]Some applications might even benefit from different versions of the midpoint operator. A dichotomic process, for example, might take advantage of using a geometric mean for wildly unbalanced intervals, while resorting to the "regular" definition otherwise.

Table VII. Algorithm to compute the midpoint of an interval according to the specifications of the draft of the future IEEE 1788 Standard on Interval Arithmetic.

```
def midpoint([a,b]):
   if [a, b] is empty:
      return NaN                # mid(empty set) == NaN
   if a == -inf:
      if b == +inf:
         return 0.0             # mid([-oo, +oo]) == 0
      else:
         return -realmax        # mid([-oo, b] == -realmax
   if b == +inf:
      return +realmax           # mid([a, +oo]) == +realmax

   round_nearest()              # Set rounding to nearest-even
   mid = 0.5*(a+b)
   if mid == -inf or mid == +inf: # Overflow in computing mid?
      return 0.5*a + 0.5*b        # Then, use Filib++'s formula
   else
      return mid                  # Else, use Boost's formula
```

In particular, not everyone agrees on what the midpoint of unbounded or semi-bounded intervals should be. For semi-bounded intervals, Arnold Neumaier [Neumaier 2010] argues for returning the other, finite, bound of the interval, his justification being that this is a sensible choice when using a midpoint in the context of centered forms [Ratschek and Rokne 1984]. On the other hand, that choice is not so useful in the context of a dichotomic exploration algorithm (Neumaier's stance on the subject is that we should use a different, specialized operator for this kind of algorithm).

For exploration algorithms, a simple choice seems to be the one advocated by Motion 37, *viz.* to return $\pm$`realmax`. Obviously, any finite value from the interval could be used too. In that situation, Zuras and Hayes [Zuras and Hayes 2012] suggest to return the midpoint of the interval obtained by replacing in the original interval the infinite bound γ by $\mathsf{sign}(\gamma) \times$ `realmax`; like the choice advocated by Motion 37, it preserves the monotonicity properties:

$$[a,b] \subseteq [a,c] \implies \mathsf{m}([a,b]) \leqslant \mathsf{m}([a,c])$$

and

$$[a,b] \subseteq [c,b] \implies \mathsf{m}([a,b]) \geqslant \mathsf{m}([c,b])$$

A downside of that approach is that it is more computationally demanding.

Even after having set the definition of the midpoint operator, there is still a lot of work to do to ensure the reliability of the actual implementation. It is not up to the IEEE P1788 standard to prescribe an implementation, as that would restrict the possibilities of implementers to take advantage of particular features of some platforms. Even after its release and widespread acceptance, there might then still be some room for some implementations to get it wrong, an easy thing to do, as this paper has shown.

ACKNOWLEDGMENTS

The origin of the work can be traced back to some email exchanges with Prof. Arnold Neumaier; Prof. Siegfried Rump read a preliminary version of this paper and gave very valuable advice and feedback, pointing out in particular an error in the first version of the proof of Prop. 3.1. Dr. Alexandre Goldsztejn

made insightful comments on an early draft of the paper. I am indebted to the ACM TOMS reviewers of this paper, whose thoroughtness, expertise and dedication helped to greatly improve its contents and presentation. All remaining errors are, naturally, my own.

REFERENCES

Alefeld, G. E. and Herzberger, J. 1983. *Introduction to Interval Computations*. Academic Press.

Brönnimann, H., Melquiond, G., and Pion, S. 2006. The design of the Boost interval arithmetic library. *Theoretical Computer Science 351,* 1, 111–118.

Chabert, G., Araya, I., Neveu, B., Jaulin, L., Trombettoni, G., and Baire, A. 2012. The IBEX constraint solver. http://www.emn.fr/z-info/ibex/.

Fog, A. 2012. The microarchitectures of Intel, AMD and VIA CPUs. An optimization guide for assembly programmers and compiler makers. Technical report, Copenhagen University College of Engineering.

Forsythe, G. E. 1966. How do you solve a quadratic equation? Technical Report CS40, Computer Science Department, Stanford University.

Forsythe, G. E. 1970. Pitfalls in computation, or why a math book isn't enough. *The American Mathematical Monthly 77,* 9, 931–956.

Goldberg, D. 1991. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys 23,* 1, 5–48.

Goualard, F. 2010. *GAOL 4.0RC: Not Just Another Interval Arithmetic Library* 5.0 Ed. Laboratoire d'Informatique de Nantes-Atlantique. http://sourceforge.net/projects/gaol.

Granvilliers, L. and Benhamou, F. 2006. Algorithm 852: Realpaver: an interval solver using constraint satisfaction techniques. *ACM Trans. Math. Softw. 32,* 1, 138–156.

Hammer, R., Ratz, D., Kulisch, U., and Hocks, M. 1997. *C++ Toolbox for Verified Scientific Computing I: Basic Numerical Problems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Hauser, J. R. 1996. Handling floating-point exceptions in numeric programs. *ACM Transactions on Programming Languages and Systems 18,* 2, 139–174.

Hayes, B. 2003. A lucid interval. *American Scientist 91,* 6, 484–488.

Higham, N. J. 2002. *Accuracy and Stability of Numerical Algorithms* Second Ed. Society for Industrial and Applied Mathematics.

Hofschuster, W. and Krämer, W. 1998. Fi_lib, eine schnelle und portable funktionsbibliothek für reelle argumente und reelle intervalle im ieee-double-format. Tech. Rep. 98/7, Instituts für Wissenschaftliches Rechnen und Mathematische Modellbildung (IWRMM) an der Universität Karlsruhe.

IEEE. 1985. IEEE standard for binary floating-point arithmetic. Tech. Rep. IEEE Std 754-1985, Institute of Electrical and Electronics Engineers. Reaffirmed 1990.

IEEE. 2008. IEEE standard for floating-point arithmetic. IEEE Standard IEEE Std 754-2008, IEEE Computer Society.

IEEE 1788 2013. IEEE interval standard working group — p1788. http://grouper.ieee.org/groups/1788/.

Intel. 2007. Intel 64 and IA-32 architectures software developer's manual: Vol. 1, basic architecture. Manual 253665-025US, Intel Corporation.

Kearfott, R. B. 1990. Preconditioners for the interval Gauss–Seidel method. *SIAM Journal on Numerical Analysis 27,* 3, 804–822.

Knüppel, O. 1994. PROFIL/BIAS—a fast interval library. *Computing 53*, 277–287.

Lerch, M., Tischler, G., Gudenberg, J. W. V., Hofschuster, W., and Krämer, W. 2006. Filib++, a fast interval library supporting containment computations. *ACM Trans. Math. Softw 32*, 299–324.

Moore, R. E. 1966. *Interval Analysis*. Prentice-Hall, Englewood Cliff, NJ.

Neumaier, A. 1990. *Interval methods for systems of equations*. Encyclopedia of Mathematics and its Applications Series, vol. 37. Cambridge University Press.

Neumaier, A. 2010. Private communication.

Ratschek, H. 1980. Centered forms. *SIAM Journal on Numerical Analysis 17,* 5, 656–662.

Ratschek, H. and Rokne, J. 1984. *Computer Methods for the Range of Functions*. Mathematics and its applications. Ellis Horwood Ltd.

Rump, S. 1999a. Fast and parallel interval arithmetic. *BIT Numerical Mathematics 39,* 3, 534–554.

Rump, S. 1999b. INTLAB - INTerval LABoratory. In *Developments in Reliable Computing*, T. Csendes, Ed. Kluwer Academic Publishers, Dordrecht, 77–104. http://www.ti3.tu-harburg.de/rump/.

Sterbenz, P. 1974. *Floating-point Computation*. Prentice Hall.

Zuras, D. and Hayes, N. T. 2012. Midpoint and unbounded intervals. Tech. rep., P1788 Working Group.